Atomic transactions

Suresh Podeti

2 min read · Apr 20, 2023

Listen Share More

Introduction

Transaction — Unit of work performed in a database system, representing a change, which can potentially composed of multiple operations.

Let T be transaction, op1, op2, op3 are operations it could be read / write operations.

```
T = {op1, op2, op3}
```

Commit a transaction — Apply changes or all operations in Transaction T.

Abort a transaction — Roll back changes applied

Databases transactions are invented in order to **simplify engineers work** and relieve them from dealing with all sort of failures. As an application developer does not need to think about scenarios of partial failures, where the transaction has failed midway after some of the operations have been performed.

It is considered atomic because either transaction gets success or failure it can be in other state.

Distributed transactions — Unit of work performed on different nodes of the distributed system, we need to ensure all nodes commit or all nodes abort. It's basically **all-or-nothing guarantee**.

In distributed system we have two kinds of distributed transactions:

1. Write **same** data into multiple nodes
2. Write **different** data on different nodes

Single object and multi-Object transactions

Single object transaction involves modifying single object (single row, document, record), where as multi-object transaction involves modify several objects (rows, documents, records). Such multi-object transactions are often needed if several pieces of the data need to be kept in sync.

Example:

Consider an example from an email application. To display the number of unread messages for a user, you could query something like this:

```
SELECT COUNT(*) FROM emails WHERE recipient_id=2 AND unread_flage=true
```

However, you might find this query to be too slow if there are many emails, and decide to store the number of unread messages in a separate field (a kind of database normalisation). Now, whenever a new message comes in, you have to increment the unread counter as well, and whenever a message is marked as read, you also have to decrement the unread counter.

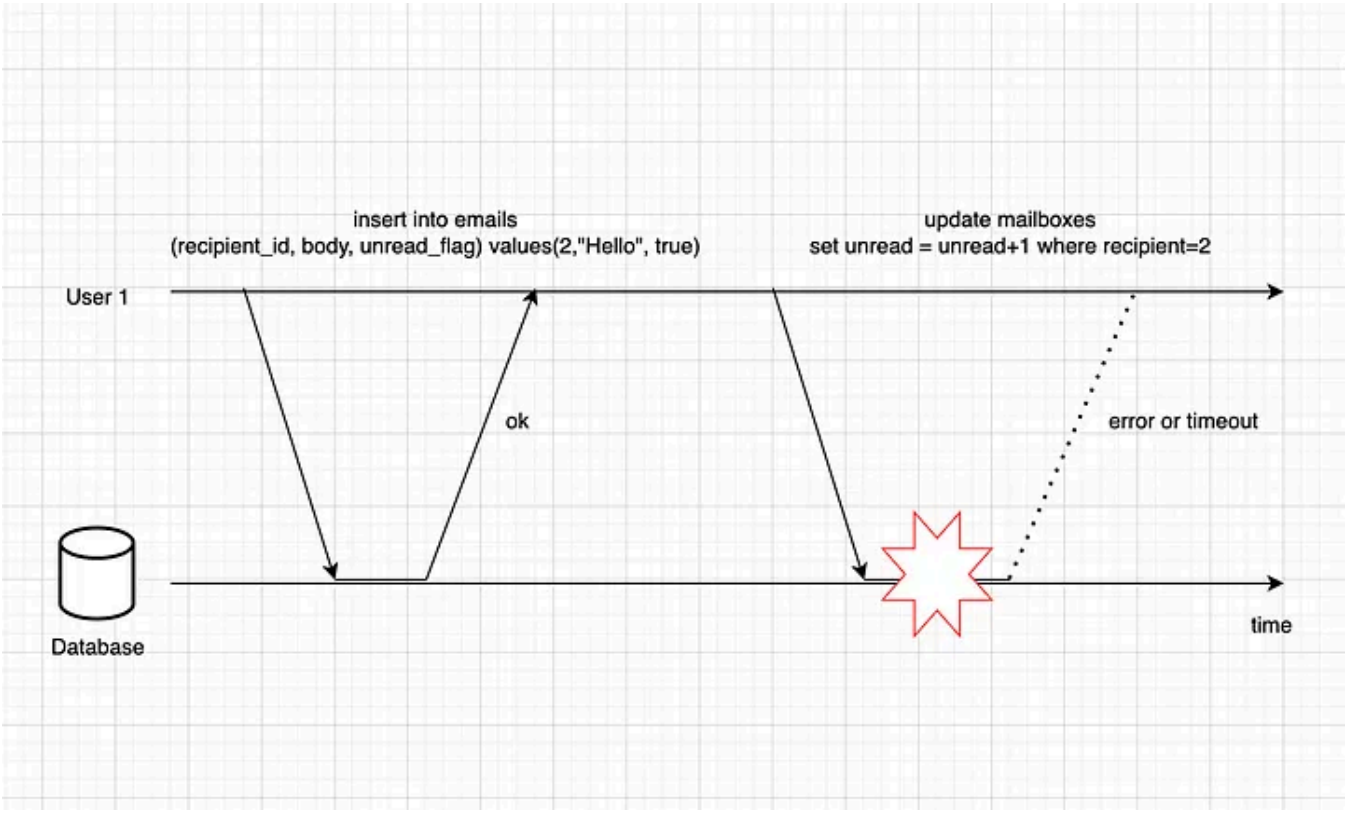

Fig 1. Atomicity ensures that if an error occurs any prior writes from that transaction are undone, to avoid inconsistent state

Figure 1 illustrates the need for atomicity: if an error occurs somewhere over the course of the transaction, the contents of the mailbox and the unread counter might becomes out of sync. In an atomic transaction, if the update to the counter fails, the transaction is aborted and the inserted email is rolled back.

In this article, we saw what is an atomic transaction and how it works.

In the next article, we will be exploring distributed transactions and some of the very popular algorithms used widely in the industry to achieve distributed transactions.

Distributed Transaction | Distributed Systems | Atomic Commit | Atomic Transaction | Atomicity

Edit profile

Written by Suresh Podeti

1.2K Followers

Aspiring Software Architect | SDE III (Golang) @JungleeGames| Ex-Byju's | Co-founder @Rupant Tech. | Ex-Synopsys | I.I.T Kharagpur | 7+ Years of Experience

Recommended from Medium